Leisure-Management-Web-App

Jakob Pirker, 1231394

12. Juni 2016

Inhaltsverzeichnis

1 Motivation und Ziele

In fast allen Vereinen und Gemeinschaften werden öfters Freizeiten angeboten. Damit sind Veranstaltungen gemeint, bei denen Mitglieder gemeinsam (meistens für mehrere Tage) an einen bestimmten Ort fahren, um dort gemeinsam Zeit zu verbringen, um einer bestimmten Tätigkeit intensiv nachzukommen (z.B. Trainingslager) oder Ähnliches. Mit der Anzahl der Teilnehmer steigt auch der Organisationsaufwand, und erreicht oft ein Maß bei dem die Organisation durch herkömmliche Methoden wie Absprache und „Papier und Stift“ unwirtschaftlich bis unmöglich wird.

Ziel dieser Arbeit ist es, eine Basis zu schaffen, die die Organisation einer solchen Freizeit vereinfacht. Dies soll dadurch geschehen, dass die im Rahmen des Projekts implementierte Anwendung eine Struktur für die Organisation vorgibt, in der oft benötigte Elemente bereits integriert sind, und direkt für die Organisation verwendet werden können. Einige Beispiele hierfür wären: Auflistung aller Teilnehmer, Überprüfung der Anwesenheiten und Veranstaltungsbeiträge jedes Teilnehmers, Aufgabenverteilungen... Die grundlegenden Anforderungen sind in den nächsten Punkten aufgelistet.

1.1 Webservice

Eine wichtige Anforderung die sich aus der Verteilung von Aufgaben vergibt ist die Bedienbarkeit von einer beliebigen Stelle aus. Aus dieser Anforderung heraus hat sich die Ausführung der Anwendung als Web-Applikation ergeben. Daraus ergeben sich folgende Vorteile:

- Bedienbarkeit von jedem internetfähigen Endgerät mit Browser
- keine Notwendigkeit spezieller Software
- zentrale Datenverwaltung (Client-Server)

1.2 Anpassbarkeit und Erweiterbarkeit

Die vorgegebene Struktur soll kein absoluter Maßstab sein, sondern der Anwender soll (wo möglich) selbst entscheiden können, welche Elemente er verwendet, und welche nicht. Außerdem soll die Applikation von einer anderen Person (mit Infromatik- Hintergrundwissen) möglichst einfach gewartet und an eine spezielle Freizeit angepasst werden können. Dies bezieht sich sowohl auf das Hinzufügen und Entfernen von Content auf der Client-Seite, als auch auf das Hinzufügen und Entfernen von zusätzlichen bzw. unnötigen Informations-Attributen und Funktionen auf der Server-Seite. Außerdem soll es möglich sein größere Informationselemente (z.B. Events) ohne viel Aufwand in die Datenstruktur zu integrieren.

1.3 Anforderungsübersicht

Es soll möglich sein Personen einzutragen, die Teilnehmer und/oder Mitarbeiter sein können, und für Personen Adressen festzulegen. Teilnehmer können einer Unterkunft

zugewiesen werden, und für diese kann wiederum eine Adresse festgelegt werden. Jedem Mitarbeiter können mehrere Aufgaben zugewiesen werden. Jeder Person können mehrere Zahlungen zugewiesen werden, mehrere Zahlungen können wiederum einem Zahlungsdepot zugewiesen werden.

2 Problemstellungen und Lösungsansätze

2.1 REST

Representational State Transfer (abgekürzt REST, seltener auch ReST) bezeichnet ein Programmierparadigma für verteilte Systeme, insbesondere für Webservices.[1]

REST verweist auf einige Prinzipien, die vorausgesetzt werden, damit der implementierte Service als RESTful bezeichnet werden kann. Bei den Prinzipien handelt es sich um:

- Client-Server
- Zustandslosigkeit
- Caching
- Einheitliche Schnittstelle
- Mehrschichtige Systeme
- Code on Demand (optional)

Im Zuge des Projekts wurde Wert darauf gelegt, dass das System diese Eigenschaften erfüllt.

2.2 Überblick

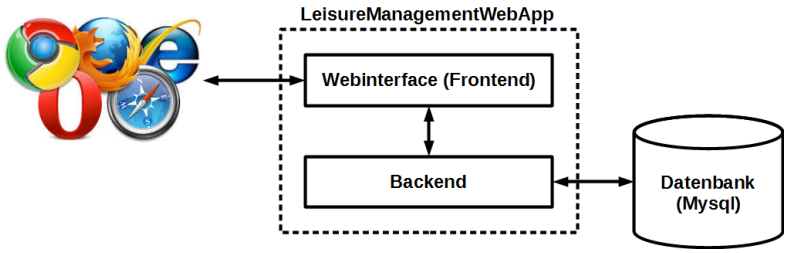

Abbildung 1: Grober Design Überblick

[1] `https://de.wikipedia.org/wiki/Representational_State_Transfer`

Wie die meisten Web-Applikationen besteht das Projekt aus einem Front- und einem Backend. Das Frontend (Client-Side) besteht aus einer Homepage, die von einem Browser aus aufgerufen wird. Es stellt eine Oberfläche zur Verfügung, mit deren Hilfe der Benutzer auf die Funktionalitäten der Applikation zugreifen kann. Das Frontend enthält jedoch nur Anzeige- und Transaktionslogik. Das bedeutet, dass es zwar Zugriff auf Funktionalitäten verschafft, und die Anzeigeelemente entsprechend der Benutzerinteraktion verändert, dann aber nur mittels eines http-Requests die entsprechenden Daten vom Backend anfordert und den zugehörigen Anzeigeelementen zuordnet. Die eigentliche Logik liegt im Backend (Server-Side). Es empfängt die Requests des Frontends, und sendet Daten als Antwort auf die Requests. So gut wie alle für die Organisation der Freizeit notwendigen Daten werden in einer Datenbank abgelegt. Das Backend fungiert auch als Schnittstelle zur Datenbank.

2.3 Backend: Spring

Das Backend wurde in Java implementiert. Hier wurde Spring[2] als unterstützendes Framework gewählt, da es einen sehr großen Funktionsumfang hat. Die beiden für dieses Projekt wichtigsten Features werden im Folgenden kurz beschrieben.

2.3.1 Datenbank Anbindung

Spring stellt mittels Spring-Data einige sehr praktische Werkzeuge zur Verfügung mit denen auf relationale Datenbanken zugegriffen werden kann. Es werden JPA-Schnittstellen zur Verfügung gestellt, die die Zuordnung und Übertragung von Objekten zu Datenbankeinträgen sehr stark vereinfachen.

2.3.2 Datenübertragung

Spring unterstützt außerdem einfache Wege, mittels denen URL's einfach auf Methoden gemappt werden können. Die empfangenen http-Requests können durch Lesen der URL-Parameter oder der Daten im http-Body in nutzbare Daten umgewandelt werden.

2.4 Frontend: AngularJS/Javascript

Zur Erstellung des Frontends wurde das JavaScript Framework AngularJS verwendet. AngularJS erweitert HTML um Attribute. Diese Attribute schaffen erstens eine sehr gute Schnittstelle zwischen den HTML-Elementen und der dahinter liegenden JavaScript Logik und zweitens ermöglichen sie eine dynamische Anpassung der HTML-Struktur des Basisdokuments.

[2] `https://spring.io`

3 Backend

3.1 Spring

3.1.1 Dependency Injection

Wie bereits erwähnt wurde als Framework für das Java-Backend Spring verwendet. Ein sehr nützliches Design-Konzept von Spring ist die Dependency Injection.

Als Dependency Injection wird in der objektorientierten Programmierung ein Entwurfsmuster bezeichnet, welches die Abhängigkeiten eines Objekts zur Laufzeit reglementiert: Benötigt ein Objekt beispielsweise bei seiner Initialisierung ein anderes Objekt, ist diese Abhängigkeit an einem zentralen Ort hinterlegt – es wird also nicht vom initialisierten Objekt selbst erzeugt.[3]

Die Abhängigkeit von einem anderen, existierenden Objekt wird im Code meistens über die `@Autowire` -Annotation gekennzeichnet. Durch Dependency Injection ist es möglich, kurzen, gut leserlichen und gut wartbaren Code zu schreiben. Einerseits weil dadurch sehr viele Dinge nicht implementiert werden müssen, andererseits weil die Komponenten der Applikation klar getrennt werden können, da die Verknüpfungen automatisch zur Laufzeit erstellt werden.

3.1.2 JavaBeans

Ein Java Grundkonzept das für das Verständnis von Spring (und bei der Fehlerbehebung während der Applikations- Entwicklung) sehr hilfreich ist, sind JavaBeans. Dabei handelt es sich im Prinzip um ein Entwurfsmuster für Klassendefinitionen. Die entworfene Klasse muss dabei folgenden Ansprüchen genügen:

- öffentlicher parameterloser Konstruktor
- Serialisierbarkeit (die Klasse ist eine Subklasse von Serializable)
- öffentliche Zugriffsmethoden (Public Getters/Setters) die einer Namenskonvention folgen

Die Vorteile von JavaBeans lassen sich am besten beschreiben durch:

Beans realisieren eine verbesserte Serialisierung und damit Netzwerkfähigkeit, Wiederverwendbarkeit, Portabilität und Interoperabilität.[4]

Damit stellen JavaBeans ein sehr gutes Konzept für die Durchführung der Dependency Injection dar, weil sie gut erzeug- und verlinkbare Objekte zur Verfügung stellen. In Spring kann die Annotation `@Component` verwendet werden, um eine Klasse als Bean zu

[3] `https://de.wikipedia.org/wiki/Dependency_Injection`
[4] `https://de.wikipedia.org/wiki/JavaBeans`

kennzeichnen. Im Code werden allerdings die Annotations `@Service`, `@Controller`, und `@Repository` verwendet, die die Klassen als Beans kennzeichnen. Diese drei Annotations sind auf einen bestimmten Zweck hin spezialisierte Ableitungen von `@Component`.

3.2 Architektur

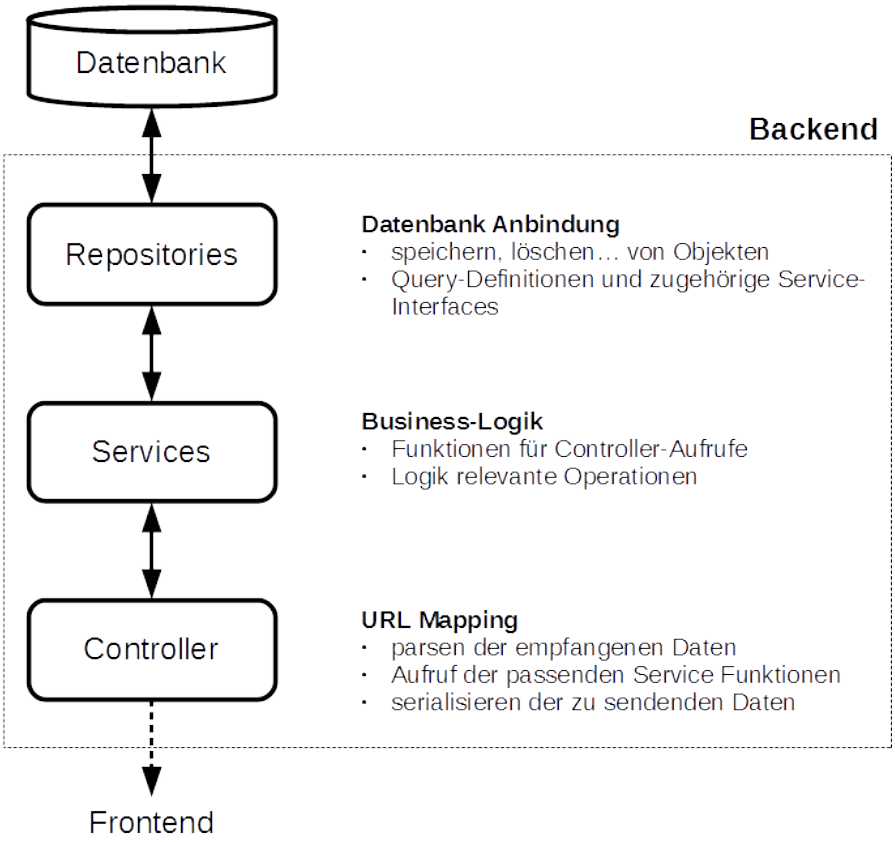

Abbildung 2: Grobes Architekturdesign des Backends

In Abbildung 2 ist eine grobe Übersicht über die Komponenten des Backends und den zugehörigen Aufgaben zu sehen. Um die Integration neuer Tabellen in das Gesamtschema zu erleichtern wurde für jede Entity-Klasse (jede Tabelle in der Datenbank) eine Klasse von jedem Architektur-Element (`Repository`, `Service`, `Controller`) erstellt. Diese Designentscheidung und die hinter jedem der Architektur-Elemente liegende Logik und Funktion wird in den folgenden Punkten genauer erläutert.

3.2.1 Entities

Die Entity-Klassen sind das Zentrum der Backend-Struktur. Da einer der wichtigsten Aspekte des Projekts Anpassbarkeit und Erweiterbarkeit ist, war es wichtig möglichst

zentrale, und klar definierte Schnittstellen dafür zu schaffen. Außerdem sollten diese Schnittstellen so unabhängig wie möglich sein, und in wenig Stellen im restlichen Code berücksichtigt werden müssen. Die wesentlichste dieser Schnittstellen ist eine Entity-Klasse. Einerseits wird hier die Verbindung und das Mapping zur Datenbank festgelegt, andererseits sind fast sämtliche Informationen für die De- und Serialisierung in für das Frontend interpretierbare Daten an dieser Stelle festgelegt. In dem Code-Segment 1 ist exemplarisch eine Entity-Klasse abgebildet, anhand der die Funktionalität der einzelnen Komponenten erklärt wird.

```java
@Entity
public class PaymentAccount {

  @Id
  @JsonProperty("Depot")
  private String id;

  @JsonProperty("Betrag")
  private Integer currentamount;

  @OneToMany(mappedBy = "paymentaccount", cascade = CascadeType.ALL)
  @JsonIgnore
  private List<Payment> payments;

  public PaymentAccount() {
    this.id = "";
    this.currentamount = 0;
  }

  public String getId() {
    return this.id;
  }
}
```

Code-Segment 1: PaymentAccount.java

Verbindung zur Datenbank

Die im Projekt verwendeten Annotations zur festlegung der JPA-Eigenschaften sind:

- `@Entity`: Markiert eine Klasse als Entity Klasse. Für jede Entity-Klasse wird eine neue Tabelle im Datenbankschema angelegt. Alle in der Klasse festgelegten Membervariablen werden auch als Einträge in der Tabelle der Datenbank erstellt. Für jeden Member wird eine Zeile erstellt. Die meisten zur Zeile zugehörigen Attribute wie z.B. Datentyp werden (sofern möglich) automatisch festgelegt. Für manche Attribute wie z.B. den Zeilen-Namen gibt es aber Wege um sie manuell festzulegen (vgl. z.B. `@Column`).

- `@Id`: Kennzeichnet ein Attribut als Primärschlüssel einer Tabelle. Jede Entity-Klasse muss mindestens einen Primärschlüssel festlegen. Es ist aber auch möglich mehrere Attribute als Primärschlüssel festzulegen. Darauf wird später noch näher eingegangen. Beim gezeigten Beispiel wird ein String, der die Bezeichnung des Payment-Accounts enthält als Primärschlüssel verwendet.

- `@OneToMany`, `@OneToOne`, `@ManyToOne`...: Damit wird eine Relation zu einer anderen Tabelle festgelegt. Die Bezeichnung der Annotation entspricht dabei der Art der Relation. Diese Annotations werden auf ein Objekt der referenzierten Klasse (= Tabelle) angewandt. Ein Payment-Account kann von mehreren Payments aus referenziert werden (weshalb es sich bei der Membervariable auch um ein `List<>`-Objekt handelt). Der Zusatz `mappedBy` drückt dabei aus, dass es sich bei dieser Seite der Relation nicht um den Owner handelt. Das bedeutet, dass die verweisenden Sekundärschlüssel in der Tabelle des anderen Objekts gespeichert werden. Über `cascade = CascadeType.ALL` wird das Framework angewiesen, Änderungen die sich auf referenzierte Objekte auswirken (in diesem Fall Payments) auch in die Datenbank zu übertragen.

- `@EmbeddedId`: Eine (und die im Projekt verwendete) Methode einen zusammengesetzten Primärschlüssel festzulegen.

- `@Column`: Explizite Deklaration einer Membervariable als Zeile in einer Tabelle. Mit Hilfe dieser Annotations können auch explizit Namen vergeben werden, auf die ein Member in der Tabelle gemappt wird.

- `@Transient`: Kennzeichnet eine Member-Variable die nicht als Zeile in die zugehörige Tabelle übernommen werden soll.

- `@GeneratedValue`: Teilt der Datenbank mit, dass die zugehörige Zeile in der Tabelle von der Datenbank selbst automatisch beschrieben wird. Diese Annotation ist für eine automatische Id-Vergabe sehr hilfreich.

Die Auswirkungen auf das tatsächliche Datenbankschema können in Abbildung 3 nachvollzogen werden.

DeSerialisierung

Für diese Aufgabe wurde die in Spring bereits integrierte Jackson-JSON-Library verwendet. Diese Library unterstützt das automatische aber gleichzeitig sehr frei definierbare umwandeln von JSON-Objekten auf POJO's (PlainOldJavaObjects). Diese Funktion erweist sich beim parsen der http-requests sehr nützlich. Dadurch können die empfangenen Daten direkt auf ein POJO gemappt, und das daraus entstehende Objekt ganz normal verwendet werden. Umgekehrt können im Laufe der Backend-Operationen erstellte oder bearbeitete Objekte wieder ganz einfach in für das Frontend verständliche JSON Objekte umgewandelt und zurückgesendet werden. Die verwendeten Annotations werden im folgenden kurz beschrieben:

- `@JsonProperty`: Kennzeichnet einen Member explizit relevant für die Umwandlung in ein JSON-Objekt. Dieser Annotation kann ein Parameter zugeteilt werden. Der wird später in das JSON-Objekt statt dem Member-Namen eingefügt. Das erweißt sich vor allem dadurch sehr nützlich, weil im Frontend dann ohne Umwege der Name des Attributes mit dem Wert für die Anzeige verwendet werden kann. Auch

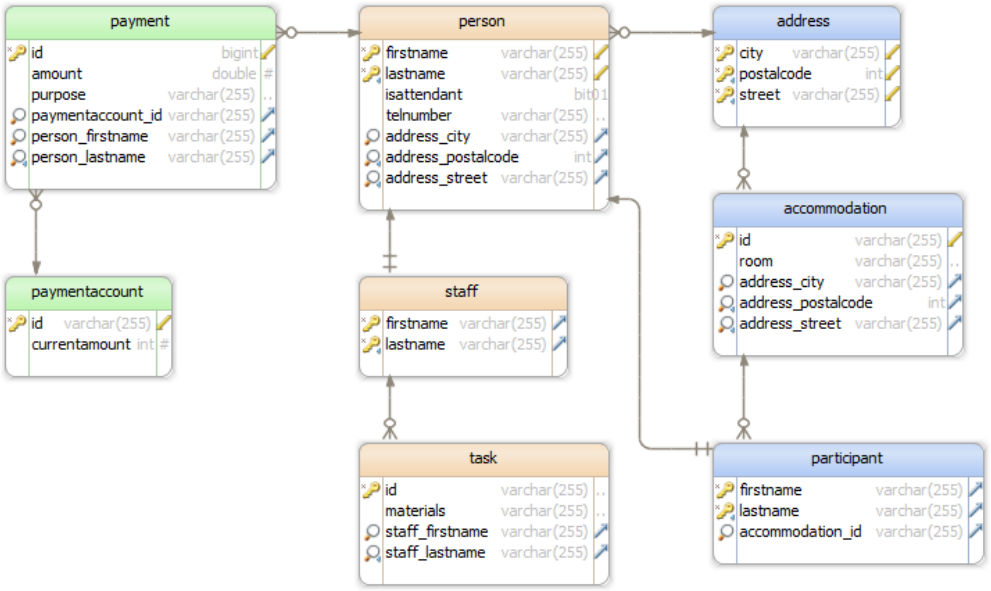

Abbildung 3: Datenbank Schema

werden die Werte von JSON-Objekten, die mit diesem Attribut Namen versehen sind wieder auf diesen Member gemappt.

- `@JsonIgnore`: Kennzeichnet eine Member-Variable als nicht relevant für die JSON-DeSerialisierung
- `@JsonUnwrapped`: Markiert ein eingebettetes Objekt, das mittels der Library DeSerialisiert werden, und die Membervariablen direkt (also nicht als eigenes Objekt) in das übergeordnete Objekt integriert werden soll.
- `@JsonPropertyOrder`: Definiert die Reihenfolge, mit der die Attribute DeSerialisiert werden sollen.

Das POJO `PaymentAccount(id = "Bankkonto", currentamount = 1351.16)` würde dem JSON `{"Depot":"Bankkonto","Betrag":1351.16}` entsprechen.

DeSerialisierung von relationalen Objekten

Eine Schwierigkeit bei der DeSerialisierung stellten Objekte dar, die über Relationen in die Klasse eingebettet sind. Erstens weil dadurch das Problem der Rekursion entsteht (Objekt A hat eine Membervariable vom Objekt B, das wiederum eine Membervariable vom Objekt C besitzt usw.). Würde diese Rekursion so umgesetzt werden, müsste ein JSON-Objekt die Informationen des Grundobjekts und aller rekursiv eingebetteten Objekte enthalten. Zweitens ist für die Verwendung eines Objektes meistens nur die Verknüpfung interessant. Es ist wichtig zu wissen, mit welchem(n) Objekt(en) Verknüpfungen bestehen, die Details der verknüpften Objekte werden aber besser an einer

anderen Stelle geklärt. Aus diesen Überlegungen heraus, hat sich die Lösung ergeben, die DeSerialisierung solcher Objekte auf die Id's derselben zu reduzieren, da die Id das fremde Objekt eindeutig definiert. Diese Funktionalität wurde mit Hilfe von `@JsonProperty`, das auf Getter und Setter der Foreign-Id angewendet wurde, realisiert. Ein Beispiel hierfür ist im Code-Segment 2 zu sehen.

```
1   @EmbeddedId
2   @JsonUnwrapped
3   private PersonId id;
4
5   /* --- content omitted --- */
6
7   @JsonProperty("Adresse")
8   public AddressId getAddress() {
9       if(this.address != null)
10      {
11        return this.address.getId();
12      }
13      else
14      {
15        return null;
16      }
17  }
18
19  @JsonProperty("Adresse")
20  public void setAddressId(String id_string) throws Exception {
21      if(id_string != null) {
22        this.address_id = mapper.readValue(id_string, AddressId.class);
23      }
24      else {
25        this.address_id = null;
26      }
27  }
```

Code-Segment 2: Foreign-Id mapping für die Address-Entity in Person.java

Ein JSON-Objekt der Person-Entityklasse könnte demnach wie im Code-Segment 3

```
1 {
2   "Nachname":"Pirker",
3   "Vorname":"Jakob",
4   "Telefonnummer":"123456789",
5   "Adresse":{"Strasse":"MeinBlock","Stadt":"MeineStadt","Postleitzahl":1111},
6   "Anwesenheit":true
7 }
```

Code-Segment 3: Beispiel für ein Serialisiertes JSON-Objekt

Primärschlüssel mit mehreren Attributen

Aus der Reduzierung eines Objekts auf seine Id ergibt sich (speziell im Hinblick auf das Frontend) die zusätzliche Anforderung an die Id, dass sie vom User direkt durch lesen einem Objekt zugeordnet werden können soll. Will man zum Beispiel eine Zahlung einer Person zuordnen, so reichen Vorname und Nachname aus, um die Person eindeutig zu kennzeichnen. Gleichzeitig erfüllt die Kombination der Attribute aber auch die Haupteigenschaft einer Id: Eindeutigkeit. Würde man einer Person zum Beispiel

nur eine Zahl als Id zuordnen, wäre diese zwar eindeutig, aber ein User müsste zuerst herausfinden, um welche Person es sich dabei handelt. Diese Anforderung macht Primärschlüssel mit mehreren Attributen notwendig. Dazu gibt es zwei verschiedene Implementierungsmöglichkeiten: `@IdClass` und `@EmbeddedId`. Beide setzen eine Zwischenklasse, die als Id fungiert und von Serializable erbt voraus.

- `@IdClass`
  - Vorteile: Id-Klasse muss zwar definiert werden, scheint aber in der Entity-Klasse nicht auf. Dadurch bleibt der Code besser lesbar, weil nur bei der Implementierung darauf geachtet werden muss, dass sich die Id's der Entity-Klasse mit denen der Basisklasse decken. Anschließend braucht nur mehr die Entity-Klasse betrachtet werden.
  - Nachteile: Doppelter Code, weil für beide Klassen die gleichen Attribute definiert werden müssen. Umständliche DeSerialisierbarkeit.
- `@EmbeddedId`
  - Vorteile: Kein doppelter Code. Die Serialisierung erfolgt bei Verwendung der `@JsonUnwrapped` vollkommen automatisch. bei der Deserialisierung kann der von der Jackson-JSON-Library zur Verfügung gestellte Mapper sehr gut verwendet werden.
  - Nachteile: Für eine vollständige Übersicht über die Attribute der Entity-Klasse muss die Id-Klasse explizit mitberücksichtigt werden.

Aufgrund der Serialisierbarkeit wurde die `@EmbeddedId`-Variante gewählt. Ein Implementierungsbeispiel ist ebenfalls im Code-Segment 2 zu sehen.

3.2.2 Repositories

Repositories stellen das Verbindungsglied zur Datenbank dar. Ihre Hauptaufgabe besteht darin, Objekte aus der Datenbank zu löschen bzw. in die Datenbank zu speichern. Außerdem werden in den Repossitories Queries definiert, und Methoden für die Services zur Verfügung gestellt, über die diese auf die Query-Ergebnisse zugreifen können. Die grundlegenden Operationen wie speichern und löschen werden bereits von einer fertigen Basisklasse `CrudRepository<>` zur Verfügung gestellt. Dem Template muss nur noch ein Parameter für den Entity-Typ sowie den Id-Typ gegeben werden. Diese Klasse stellt auch schon einige wichtige Query Methoden zur Verfügung (ein oder mehrere Objekte Anhand der Id auslesen und alle Objekte auslesen). Es ist in Spring nicht notwendig, die Repository-Interfaces explizit in eigenen Klassen zu implementieren. Spring stellt aus einer Interface Definition für ein Repository automatisch die dazugehörige Implementierung her.
Ein sehr praktisches Feature dabei sind die sogenannten Named-Queries. Folgt der Methoden-Name in einem Repository-Interface einer bestimmten Form, so kann Spring aus dem Namen die zugehörige Query selbst bestimmen. Somit kann Query- und Methoden-Definition in einem Zug erledigt werden, was die Wartbarkeit, Lesbarkeit und Änderbarkeit

des Codes sehr verbessert. Der Methodenname setzt sich dabei aus den Membernamen der zugehörigen Entity-Klasse und aus Keywords zusammen, die es ermöglichen die Member logisch zu verknüpfen. Ein Beispiel für eine mögliche Named-Query im `PersonRepository` wäre:

```
1   List<Person> findByLastnameAndIsattendantOrderByLastnameAsc(String lastname,
      Boolean isattendant);
```

Mit dieser Query würde man aus der Person-Tabelle die Personen mit den in den Funktionsparametern gegebenen Werten für `lastname` und `isattendant` herausfiltern, und sie aufsteigend nach dem Nachnamen sortiert bekommen.
Weil oft für alle oder mehrere Objekte die gleichen Queries definiert werden sollen wurde ein Basis-Interface erstellt, das diese gemeinsamen Methoden enthält. Die Implementierung ist in Code-Segment 4 zu sehen. Das Interface ist von `CrudRepository<>` abgeleitet, wodurch die endgültigen Repositories nur mehr von `AbstractRepository<>` abgeleitet werden müssen. Außerdem ist `AbstractRepository<>` mit der Annotation `@NoRepositoryBean` versehen. Dadurch wird für dieses Interface keine eigene Klasse erzeugt, was auch gar nicht möglich wäre, da es sich um ein Template handelt. Die abgeleiteten Repositories können natürlich wieder eigene Named-Queries definieren, bieten aber auch gleichzeitig die Methoden von `AbstractRepository<>` an. Die Implementierung eines abgeleiteten Interfaces ist in Code-Segment 5 zu sehen.

```
1 @NoRepositoryBean
2 public interface AbstractRepository<T_obj, T_id extends Serializable> extends
      CrudRepository<T_obj, T_id> {
3   Iterable<T_obj> findAllByOrderByIdAsc();
4 }
```

Code-Segment 4: AbstractRepository.java

```
1 public interface PersonRepository extends AbstractRepository<Person, PersonId> {
2 }
```

Code-Segment 5: PersonRepository.java

3.2.3 Controller

Wie bereits erwähnt existiert für jede Entity-Klasse u.A. auch eine Controller-Klasse. Sie stellt die Schnittstelle zum Frontend dar, indem URL-Aufrufe auf bestimmte Funktionsaufrufe gemappt werden. Jede Entity-Klasse soll im Endeffekt über das Frontend bearbeitbar sein. Für jedes Objekt soll es dabei einen Grundsatz an Funktionen geben, die für jede Entity verfügbar ist. In der `AbstractController<>` (vgl. Code-Segment 8) Klasse wird genau dieser Grundsatz definiert. Dabei werden folgende Requests folgenden Responses zugeordnet:

- **GET-Request** auf Basis-URL: String, der die Struktur der Entity beschreibt. Dieser Teil wird im Rahmen der Services noch genauer beschrieben.
- **POST-Request** auf Basis-URL: Speichern eines neuen Objekts in die Datenbank. Das neue, zu speichernde Objekt wird dabei über `@RequestBody` aus den Daten des

http-Requests extrahiert. Die Daten werden Dabei in Form eines JSON-Objektes erwartet, `@RequestBody` mappt dieses JSON-Objekt direkt auf das in der Argumentliste. definierte POJO. Als Antwort auf diesen Request wird das gespeicherte Objekt gesendet. Dieses wird implizit über die Jackson-JSON-Library in ein JSON-Objekt umgewandelt, das in die Daten der http-Response geschrieben wird.

- **DELETE-Request** auf Basis-URL: Löschen eines Objektes aus der Datenbank. Das zu löschende Objekt wird gleich wie beim POST-Request aus den Daten extrahiert.
- **GET-Request** auf Basis-URL/list: Liste von allen Objekten des vorgegebenen Typs in der Datenbank. Auch hier wird die erzeugte Liste wie beim POST-Request implizit in ein JSON-Objekt serialisiert, und in den Daten der http-Response gesendet.

Für einen konkreten Controller muss dann nur (wie in Code-Segment 7 zu sehen) eine Klasse erstellt werden, die von dieser Basisklasse erbt. Wichtig ist hier auch noch, über `@RequestMapping` den URL-Pfad, über den die Entität verfügbar ist zu mappen. Auch wichtig ist hier, dass die Klasse mit `@RestController` gekennzeichnet wird. Damit wird wie schon beschrieben eine JavaBean erzeugt, die mittels `@Autowired` automatisch zugewiesen werden kann. Bei der abstrakten `AbstractController<>` Klasse muss diese Annotaion natürlich weggelassen werden, da es sich a) um eine abstrakte Klasse und b) um ein Template handelt.

```
1  public abstract class AbstractController<T_obj extends Object, T_serv extends
       AbstractServiceInterface<T_obj>> {
2
3    @Autowired
4    protected T_serv service_;
5
6    @RequestMapping(method = RequestMethod.GET)
7    public String getRequest() {
8      return service_.getJsonStringWithForeignIds();
9    }
10
11   @RequestMapping(method = RequestMethod.POST)
12   public T_obj postRequest(@RequestBody T_obj new_obj) {
13     return this.service_.save(new_obj);
14   }
15
16   @RequestMapping(method = RequestMethod.DELETE)
17   public void deleteRequest(@RequestBody T_obj delete_obj) {
18     this.service_.delete(delete_obj);
19   }
20
21   @RequestMapping(value = "/list", method = RequestMethod.GET)
22   public Iterable<T_obj> getList() {
23     return service_.getList();
24   }
25 }
```

Code-Segment 6: AbstractController.java

```java
@RestController
@RequestMapping("/person")
public class PersonController extends AbstractController<Person, PersonService>{
}
```

Code-Segment 7: PersonController.java

```java
public abstract class AbstractController<T_obj extends Object, T_serv extends
    AbstractServiceInterface<T_obj>> {

@Autowired
protected T_serv service_;

@RequestMapping(method = RequestMethod.GET)
public String getRequest() {
return service_.getJsonStringWithForeignIds();
}

@RequestMapping(method = RequestMethod.POST)
public T_obj postRequest(@RequestBody T_obj new_obj) {
return this.service_.save(new_obj);
}

@RequestMapping(method = RequestMethod.DELETE)
public void deleteRequest(@RequestBody T_obj delete_obj) {
this.service_.delete(delete_obj);
}

@RequestMapping(value = "/list", method = RequestMethod.GET)
public Iterable<T_obj> getList() {
return service_.getList();
}
}
```

Code-Segment 8: AbstractController.java

Ein GET-Request auf die URL `http://localhost:8080/person/list` könnte folgende Response-Daten liefern:

```
[
  {
  "Nachname":"Pirker",
  "Vorname":"Jakob",
  "Telefonnummer":"123456789",
  "Adresse":{"Strasse":"MeinBlock","Stadt":"MeineStadt","Postleitzahl":1111},
  "Anwesenheit":true
  },
  {
  "Nachname":"Mustermann",
  "Vorname":"Max",
  "Telefonnummer":"987654321",
  "Adresse":{"Strasse":"AuchMeinBlock","Stadt":"AuchMeineStadt","Postleitzahl
    ":2222},
  "Anwesenheit":false
  }
]
```

3.2.4 Services

Der Service Stellt im Prinzip die Verbindung zwischen Repositories und Controllern her. Natürlich können hier auch Anwendungsspezifische Erweiterungen in der Logik imple-

mentiert werden. Um die grundsätzliche Funktionalität möglichst einfach zu gewährleisten wurde auch hier eine abstrakte Template - Klasse erzeugt, die die Standardoperationen nativ unterstützt (s. Code-Segment 9).

```
1  public abstract class AbstractService<T_obj, T_id extends Serializable, T_rep
       extends AbstractRepository<T_obj, T_id>> implements AbstractServiceInterface<
       T_obj>{
2
3    @Autowired
4    protected T_rep base_repository_;
5
6    protected ObjectMapper object_mapper_ = new ObjectMapper();
7
8    public T_obj save(T_obj save_object){
9      return this.base_repository_.save(save_object);
10   }
11
12   public void delete(T_obj delete_obj) {
13     this.base_repository_.delete(delete_obj);
14   }
15
16   public Iterable<T_obj> getList(){
17     return base_repository_.findAllByOrderByIdAsc();
18   }
19 }
```

Code-Segment 9: AbstractService.java

Grundsätzlich kann diese Basisklasse wie auch schon beim Controller direkt verwendet werden. Ein wesentlicher Punkt, den sie aber nicht automatisiert beherrscht ist der Umgang mit Fremdschlüsseln. Dieser muss für jede Entität die Fremdschlüssel enthält eigens festgelegt werden. Ein Beispiel hierfür ist in Code-Segment 10 anhand des PersonService zu sehen.

```
1  @Service
2  public class PersonService extends AbstractService<Person, PersonId,
       PersonRepository>{
3
4    @Autowired
5    private AddressRepository address_repository_;
6
7    @Override
8    public Person save(Person save_person){
9
10     // try to insert the valid address-object from the DB by it's id (from JSON)
11     if(save_person.getAddress() == null && save_person.getAddressId() != null)
12     {
13       save_person.setAddress(address_repository_.findOne(save_person.getAddressId
       ()));
14     }
15       return this.base_repository_.save(save_person);
16     }
17
18   public String getJsonStringWithForeignIds() {
19
20     JsonNode person = object_mapper_.valueToTree(new Person());
21     JsonNode address_ids = object_mapper_.createArrayNode();
22
23     Iterable<Address> addresses = address_repository_.findAll();
24
25     for(Iterator<Address> i = addresses.iterator(); i.hasNext(); ) {
```

```
26        ((ArrayNode) address_ids).add(object_mapper_.valueToTree(i.next().getId()));
27      }
28
29      ((ObjectNode) person).put("Adresse", address_ids);
30
31      return person.toString();
32    }
33  }
```

Code-Segment 10: PersonService.java

Beim speichern müssen die Fremdschlüssel berücksichtigt werden. Da Objekte bei der Serialisierung wie in Abschnitt 3.2.1 bereits erläutert nur die Id hinterlegt wird, muss (wie in `save()` zu sehen) beim speichern das betreffende Objekt noch richtig gesetzt werden (die Speicherung folgt dann über die auch in 3.2.1 beschriebene Kaskadierung automatisch).
Auch beim der Beschreibung der Entity-Struktur müssen die Fremdschlüssel berücksichtigt werden. Einerseits muss dem Frontend irgendwie bekannt gemacht werden, dass es sich bei diesem Objekt um einen Fremdschlüssel handelt, andererseits wäre es sehr wünschenswert, wenn die verfügbaren Fremdschlüssel bei der Erstellung eines neuen Entity-Objekts gleich verfügbar wären, um eine leichte Auswahl zu ermöglichen. Das wird genau in der Methode `getJsonStringWithForeignIds()` realisiert. Es wird einfach ein neues Objekt vom geforderten Typ erzeugt, und in ein JSON-Objekt umgewandelt. Die verfügbaren Fremdschlüssel werden dann nachträglich als Liste (wieder als JSON-Objekte) in das leere Objekt, statt dem leeren Verweis auf einen Fremdschlüssel eingefügt.

4 Frontend

4.1 Angular JS

4.1.1 Zwei-Weg Datenbindung

4.2 Tabs

4.3 Tabellen

4.4 Eingabeformen

5 Installation und Anmerkungen

6 Quellen

- REST: https://de.wikipedia.org/wiki/Representational_State_Transfer
- REST: https://spring.io/understanding/REST
- JPA: https://de.wikipedia.org/wiki/Java_Persistence_API

6.1 Spring

- https://spring.io
- Dependency Injection: https://de.wikipedia.org/wiki/Dependency_Injection
- JavaBeans: https://de.wikipedia.org/wiki/JavaBeans
- Beans Annotations: http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/beans.html#beans-stereotype-annotations
- Repositories: http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-property-expressions

6.2 AngularJS

- https://de.wikipedia.org/wiki/AngularJS
- https://angularjs.org/
- http://www.w3schools.com/angular/

6.3 Abbildungen

- Abbildung 1 (Browser Zusammenstellung): http://vestavialibrary.org/wp-content/uploads/2016/02/web-browsers.jpg